DØ Note XXXX

Recommendations Regarding Common Analysis Format Infrastructure

D0 Data Format Working Group

October 4, 2004

Contents

1 Introduction

The Report of the DØ Data Format Working Group [1] makes a number of recommendations regarding the software infrastructure as it relates to the proposed Common Analysis Format (CAF).

1.1 Processing Chain

The processing chain from raw data to CAF trees looks as follows:

- d0reco (produces thumbnails).
- Fixing of thumbnails (if required for a given reco release).
- Skimming of thumbnails.
- Generation of CAF trees.
- Skimming of CAF trees.
- User analysis of CAF trees (producing histograms, plots etc.)

Optionally, the the generation of CAF trees may already include further skimming and the generation of more specialized output streams.

In general the event selection after the CAF tree generation is left to physics groups or individuals. However, common tools will be provided to do this in a coherent way and to generate any necessary metadata to store these samples back into SAM if necessary.

1.2 Use Cases

The following are the most common use case concerning the processing chain.

Common CAF tree generation The standard version of the CAF tree is produced centrally from the corresponding thumbnail skims. It is assumed that these files are stored back into SAM to make them easily available to the whole collaboration.

Extended CAF tree generation by user/group A group or user may generate CAF trees with extended information which is not available in the standard format. The same tools should be used for this as for the normal CAF trees, to avoid a proliferation of incompatible trees.

Skimming of CAF trees by user/group A user or group applies a first step of event selection for her analysis and skims the central CAF trees further down. It should be possible to store it back into SAM, depending on the size of the resulting sample and how many people need to access it.

Storing of CAF trees in SAM A user or group has a data sample that will be accessed by other people. It should be stored back into SAM.

Accessing CAF trees from SAM for user analysis Any user starting from the common CAF tree skims or private ones will access root trees from SAM.

2 Generation of CAF Files

In the following we outline the generation procedure of CAF trees and the requirements on the generation program.

2.1 Overview of tmb_analyze

`tmb_analyze` is the DØ package that provides the `TMBAnalyze_x` executable. It is currently used to generate TMBTrees. It's internals are outlined in [2].

It basically consists of a set of subclasses of `TMBMaker`, each of which is responsible for handling a specific D0 EDM chunk and transforming it into a ROOT tree branch. `Makers` are registered by various DØ framework packages which allows a modular design.

There are several short-comings of the current implementation of `TMBAnalyze_x`.

- It doesn't generate correct metadata for SAM.
- It allows only one output file specification. If a file grows too big, ROOT will automatically open a new one, but without respecting the original input file boundaries. This make it impossible to generate the proper parentage information for SAM.
- It does not allow a selection of events based on tags. At the moment the only selection possible is to filter events. This makes it impossible to use the executable with the standard skimming package `np_tmb_stream` and produce multiple TMBTree files in parallel with different selection criteria.

2.2 Requirements

The above short-comings should be addressed on the time-scale of the project. Especially, these are:

SAM Metadata The program should produce proper SAM metadata. It should use the standard `sam_manager` to do this. Application names and versions should be registered with SAM, so users can select files based on these criteria. The correct data tier (`root-tree` or `root-tree-bygroup`) should be produced.

Multiple Output Streams The program should be able to create multiple output files with different event selection criteria and without running the `TMBMaker` code multiple times on the same event.

Multiple Output Files If the output for a given stream requires more than one output file, it shall be possible to preserve input file boundaries (like `WriteEvent` does).

Tagged Event Selection The program should be able to select events based on tags, as produced by `np_tmb_stream`

For the SAM metadata we will use different applications versions for central production and for other users/groups, to keep the output clearly separated. An application family `treemaker` and an application name `tmb_analyze` as well as a data tier `root-tree` and `root-tree-bygroup` already exist.

3 Analysis Environment

3.1 Current Practice

Given the multitude of root based formats there is a wide variety of ways people access the data for their analysis. We try to summarize them here, with an emphasis on TMBTree and top tree.

Interactive ROOT Use Most tuple based formats can be simply opened in ROOT and displayed with the `TTree::Draw()` method. This is also true for object oriented ROOT formats, with the caveat that values that are recalculated by methods are not available without the original library.

MakeClass & friends `MakeClass` allows you to generate a simple analysis program based on the content of a ROOT file. It works for tuple based formats, but obviously cannot regenerate any methods for an object based format. Newer versions likes `MakeProxy` and `MakeProject` are able to regenerate most of the object structure, again without any user defined methods.

Macros to recompile the TMBTree Library The `tmb_analyze` package contains ROOT macros to recompile all needed TMBTree classes into a shared library that can be used with ROOT. This makes all the original methods available to the user. Analysis code that uses this library can also be compiled with `ACLiC`, the ROOT compilation system.

Custom Makefiles Various `Makefiles` are available that compile the TMBTree classes into a ROOT shared library. It is often easy to extend the Makefiles to add any user specific code.

`top_tree_reader` Packages like `top_tree_reader` provide a standardized way to read top trees and loop over the events.

Frameworks Multiple groups (typically with many analyzers working together) created a framework where different users can plug in their code in a standardized way. The framework usually takes care of reading events and providing all the necessary libraries in a pre-compiled form. Often it contains a well defined interface to access data (e.g. via an `Event` class).

3.2 Requirements

We feel that there are different ways that people would like to access the data in a CAF tree, depending on the situation. The CAF environment should provide the tools to do this with minimal effort, without forcing the user to use a full-fledged framework when all she wants to do is a having a quick look at a few distributions.

On the other hand, users should not be required to re-invent their own framework, if it is necessary to share code easily with others. Ideally, the CAF environment consists of building blocks which can be used in an interactive environment as well as in a framework, and code written against these interfaces will work in both.

3.3 Interactive Analysis

By interactive analysis we refer to the common task of opening a ROOT file and quickly plotting some variables. The `TTree::Draw()` method of ROOT is powerful enough to allow many common selections and allows the easy creation of histograms, which can subsequently stored in a file.

Since the CAF format will not store all variables in the tree, but recalculate them where possible, it requires the presence of a corresponding shared library which contains all the methods. Creating this library is now done in a multitude of ways.

3.3.1 Working in a D0 Release

The latest versions of the D0 release produce shared libraries in addition to the usual static libraries. When working in a D0 release, one can therefore directly use the shared library from the release.

```
> setup D0RunII p17.00.00
> root
root[0] .L $SRT_PUBLIC_CONTEXT/shlib/$SRT_SUBDIR/libmet_util.so
root[1] .L $SRT_PUBLIC_CONTEXT/shlib/$SRT_SUBDIR/libtmb_tree.so
```

The latest version of ROOT included in the p17 release also provides an auto-loading feature that can be used to make the above steps transparent. The following command creates a file in the current working directory that tells ROOT where to find the shared libraries needed for a given class. When using that class in the interpreter, ROOT will automatically load it.

```
> rlibmap -r .rootmap -l \
  $SRT_PUBLIC_CONTEXT/shlib/$SRT_SUBDIR/libtmb_tree.so \
  -d $SRT_PUBLIC_CONTEXT/shlib/$SRT_SUBDIR/libmet_util.so \
  -c $SRT_PUBLIC_CONTEXT/tmb_tree/src/*_linkdef.h

> root
root[0] TMBMuon m;
```

The above will “just work” in the interpreter, without requiring the user to do anything in addition.

3.3.2 Working outside a D0 Release

Some people prefer to have only the shared library for the TMB trees and work with a ROOT version not coupled to any D0 release, e.g. on a laptop or a machine where no D0 software is installed.

Either a common ROOT macro or a portable Makefile should be provided centrally to build the shared library from scratch. This should not assume the existence of any D0 software environment. Ideally it should

- Work with any ROOT version and take the compiler flags e.g. from `root-config`.
- Work on the original package structure (i.e. will not require the user to copy all header and source files to a different place)
- Work on any additional packages that the user writes in the same way.

3.4 Analysis Using the ROOT Infrastructure

ROOT provides the equivalent of an event loop, based on the `TTree::Process()` method. It expects an object of type `TSelector`. Users typically derive from that base class and implement methods which are called at the appropriate time (beginning of processing, change of input tree, next event, end of processing etc.).

Advantages of using `TSelector` are

- Works transparently with `TTree` and `TChain`.
- Works transparently with PROOF.

`TChain` will not work with SAM, since it requires the existence of all files in the chain. There is no implementation of a `SAMTree` class with functionality similar to `TChain` for a SAM dataset.

Only a single `TSelector` can be passed to the `TTree::Process()` method. Code written for this interface can therefore not be easily modularized (e.g. how do you run two `TSelector` objects in sequence) except with an additional level of indirection.

3.5 Root Framework

There are various ROOT based frameworks in existence in DØṠome are widely used like `d0root`, others were developed and used by only a small handful of users.

`d0root` has been initially developed to try to overcome the problem of incompatible data formats. It does this by converting all the different inputs into yet another internal data format.

Assuming that the CAF tree represents the common DØ ROOT format for analysis, a common framework should build directly on this interface. This avoids any additional conversion and copying that is unavoidable with the `d0root` approach.

The following requirements and suggestions are taken from the DFWG report, presentations in the DFWG meetings about existing frameworks and internal discussions.

3.5.1 Event Class

The equivalent of the Event Data Model (EDM) in ROOT is based on the notion of different *branches* in a `TTree` structure. A tree can contain an unlimited number of branches, each with a different internal structure. This makes the tree a flexible and extensible data structure. Access to a given branch is achieved by passing the tree the address of an object of the appropriate type. For each event the `TTree` will fill the user object with the data from the file.

For each branch the user has to know the correct type and its name. Once the address is set, it has to be passed around to the user code (or retrieved from the tree itself).

Almost all frameworks in use provide an *Event* object which centralizes the access to the data. It is the *Event* object which is filled once and then passed to the user code. Every user routine is just called with a reference to the current event.

The code in the *Event* class is responsible for setting up the branches and addresses for the various physics objects. It should provide the user with a type-safe way of accessing all the common objects she needs for her analysis. Whenever possible, this should happen with no additional copying taking place.

Expert users might create CAF trees with non-standard content. Users might calculate some derived quantities from the original CAF tree which they want to store in ROOT files and access later. In both cases it should be possible to access these data via the common *Event* class without changing the *Event* class itself.

It is often desirable to calculate quantities in one place and then make them available for later uses. The *Event* class should also serve as an intermediate store for such data. This is to avoid the use of global variables. It should be possible to store arbitrary data in the *Event* without the requirement to inherit from a base class etc. For example, it should be possible to store a single integer value without much additional overhead.

3.5.2 Root Packages

The framework should provide the equivalent of a simplified DØ package in ROOT. Each package should do one simple thing and the framework should put them together in a user defined way.

ROOT packages should be much simpler to write and develep than framework packages. For example, it should be possible to use a simple function as a framework package. Instead of providing an unlimited number of hooks and interfaces, the ROOT framework should restrict itself to the most needed tasks: information about beginning and end of processing, change of input files, and a process hook. The process hook can return a boolean indicating if this event should be further processed, combining both the `processEvent` and `filterEvent` interface of the DØ framework.

With the existence of shared libraries in the standard DØ release, the framework should make best use of them. For example, if packages follow a well-defined structure, it is possible to load them at run-time without explicit linking at compile time. This should speed up the developement time and require the existence of only a single executable (and not one for each analysis).

ROOT packages should not be concerned with issues related to input of event data. It should be transparent if the data comes from a single file, a list of files, from SAM etc. However, there should be a mechanism to allow for partial reading of events. This has been shown to speed up I/O remarkedly.

3.5.3 Configuration Mechanism

A simple mechanism should be provided to configure the ROOT packages. Code for this packages should be parameterized via the configuration mechanism instead of hard coding values.

ROOT uses a simple file syntax (see the `TEnv` class) for its own configuration files. This provides lookup of name/value pairs. Parameters of an analysis are expected to change over time and not be constant. The use of a database oriented approach like RCP seems therefore not appropriate. Using the ROOT based file syntax allows immediate access to these configuration files without much additional coding effort.

3.6 SAM Interface

The ROOT framework should hide all details concerning the input of event data. Especially, it should be transparent if the files are sitting on a local or if they are delivered by SAM.

The `sam_client_api` package allows access to SAM from ROOT. It has been tested on CAB and ClueDØ both with single and parallel jobs. It expects that the SAM project is setup outside of the library, however, this is usually the case with parallel jobs anyway.

3.7 Batch Framework

Naturally, only executables that have been built inside the DØ framework should be expected to run in the batch system. Since these are similar to a standard DØ framework executable, `d0tools` should be adapted to handle any differences which remain.

3.8 Root Build System

With the latest p17 release, the (still optional) use of shared libraries has sped up the turn-around time for developing DØ code. We therefore suggest to do all normal developement inside the usual DØ release structure.

The following optimizations have been suggested:

- Have a special *analysis* release that contains only the few packages relevant for a ROOT based analysis. This mainly speeds up the linking process which is now often the most time-consuming part. The reason for this is that the linker will search through a symbol database of all libraries in the DØ release. Reducing the number of packages will make this step shorter. However, with the use of run-time loading, the linking step might be mostly obsolete.

- Have a stand-alone build system which can be used outside the DØ environment. This would be either based on the built-in ROOT ACLiC system, or custom Makefiles.

4 Documentation System

References

[1] D0 Note 4473, Report of the DØ Data Format Working Group.

[2] D0 Note xxxx, Recommendations Regarding Common Analysis Format Content.